SANDIA REPORT

SAND2004-3268 (Updated Novenber 2008)
Unlimited Release
Printed November 2008

Teuchos::RCP Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Department of Optimization and Uncertainty Estimation

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2004-3268 (Updated Novenber
2008)
Unlimited Release
Printed November 2008

Teuchos::RCP Beginner's Guide

An Introduction to the Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++

Roscoe A. Bartlett
Optimization and Uncertainty Estimation
Sandia National Laboratories [*], Albuquerque NM 87185 USA,

Abstract

Dynamic memory management in C++ is one of the most common areas of difficulty and errors for amateur and expert C++ developers alike. The improper use of operator `new` and operator `delete` is arguably the most common cause of incorrect program behavior and segmentation faults in C++ programs. Here we introduce a templated concrete C++ class `Teuchos::RCP`, which is part of the Trilinos tools package `Teuchos`, that combines the concepts of smart pointers and reference counting to build a low-overhead but effective tool for simplifying dynamic memory management in C++. We discuss why the use of raw pointers for memory management, managed through explicit calls to operator `new` and operator `delete`, is so difficult to accomplish without making mistakes and how programs that use raw pointers for memory management can easily be modified to use `RCP`. In addition, explicit calls to operator `delete` is fragile and results in memory leaks in the presents of C++ exceptions. In its most basic usage, `RCP` automatically determines when operator `delete` should be called to free an object allocated with operator `new` and is not fragile in the presents of exceptions. The class also supports more sophisticated use cases as well. This document describes just the most basic usage of `RCP` to allow developers to get started using it right away. However, more detailed information on the design and advanced features of `RCP` is provided by the companion document "Teuchos::RCP : The Trilinos Smart Reference-Counted Pointer Class for (Almost) Automatic Dynamic Memory Management in C++".

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgments

The author would like to thank Carl Laird, Heidi Thornquist, Mike Heroux and Marzio Sala for comments on earlier drafts of this document.

The format of this report is based on information found in [4].

Contents

Appendix

1 Introduction

The main purpose of this document is to provide a quick-start guide on how to incorporate the reference-counting smart pointer class `Teuchos::RCP` into C++ programs that use dynamic memory allocation and object orientation. This code is included in the Trilinos [3] tools package `Teuchos`. The design of `Teuchos::RCP` is based partly on the interface for `std::auto_ptr<>` and Items 28 and 29 in "More Effective C++" [5]. In short, `RCP` allows one client to dynamically create an object (using operator `new` for instance), pass the object around to other clients that need to access the object and never require any client to explicitly call operator `delete`. The object will (almost magically) be deleted when all of the clients remove their references to the object. In principle, this is very similar to the type of garbage collection that is in languages like Perl and Java. There are some pathological cases (such as the classic problem of circular references, see [5, Item 29, page 212]) where `RCP` will result in a memory leak, but these situations can be avoided through the careful use of `RCP`. However, realizing the potential of hands-off garbage collection with `RCP` requires following some rules. These rules are partially spelled out in the form of commandments in Appendix C.

Note that direct calls to operator `delete` are discouraged in modern C++ programs that are designed to be robust in the presence of C++ exception handing. This is because the raw use of operator `delete` often results in memory leaks when exceptions are thrown. For example, in the code fragment:

```
void someFunction()
{
  A *a = new A;
  a->f();
  delete a;
}
```

if an exception is thrown in the function call `a->f()` then the statement `delete a` will never be executed and a memory leak will have been created. The class `std::auto_ptr<>` was added to the standard C++ library (see [5, Items 9 and 10]) to protect against these types of memory leaks. For example, the rewritten function:

```
void someFunction()
{
  std::auto_ptr<A> a(new A);
  a->f();
}
```

is robust in the event of exceptions and no memory leak will occur. However, `std::auto_ptr<>` can not be used to share a resource between two or more clients and therefore is not an answer to the issue of general garbage collection. The class `RCP` not only is robust in the event of exceptions but also implements reference counting and is therefore more general (but admittedly more complex and expensive) than `std::auto_ptr<>`.

The use of `RCP` is critically important in the development and maintenance of large complex object-oriented programs composed of many separately-developed pieces (such as Trilinos). This

discussion assumes that the reader has a basic familiarity and some programming experience with C++ and has at least been exposed to the basic concepts of object-oriented programming (good sources include [2] and [6]). Furthermore, the reader should be comfortable with the use of C++ pointers and references.

The appendices contain basic reference material for RCP. In many respects, the appendices are the most important contribution of this document. For those readers that like to see the C++ declarations right away, Appendix A contains the C++ declarations for the template class RCP and some important associated non-member templated functions. Appendix B is a short reference-card-like quick-start for the use of RCP. The quick-start in this appendix shows how to create RCP objects from raw C++ pointers, how to represent different forms on constantness, cast from one pointer type to another, access the underlying reference-counted object as well as to associate and manage extra data. Appendix C gives some commandments for the use of RCP and reinforces the material in Appendix B. Appendix D gives tables of recommended idioms for how to pass raw C++ objects and RCP-wrapped objects to and from functions. Appendix E gives a listing for an example program that uses raw pointer variables and direct calls to operator new and operator delete while Appendix F shows a refactoring of this example program to use RCP.

Note! Anxious readers are encouraged to jump directly to Appendix E and F to get an idea of what RCP is all about. This example, together with the reference material in the appendices, should be enough for semi-experienced C++ developers to start using RCP right away.

For less anxious readers, in the following section, we describe why the use of raw C++ pointers and raw calls to operator new and especially operator delete is difficult to program correctly in even moderately complex C++ programs. We then discuss the different ways C++ pointers are used in such programs and describe how to refactor these programs to replace some of the raw C++ pointers and raw calls to operator delete with RCP. In the following discussion we will define *persisting* and *non-persisting* associations and will make a distinction between them (see page 11). RCP is recommended for use only with *persisting* associations. The consistent use of RCP extends the vocabulary of C++ in helping to distinguish between these two types of relationships. In addition, RCP is designed for the memory management of individual objects, not raw C++ arrays of objects. Array allocation and deallocation should be performed using standard C++ containers such as std::vector<>, std::valarray<> or some other such convenient C++ array class but the best choice is typically a debug range-checked class like Teuchos::Array. However, it is quite common to dynamically allocate arrays of RCP objects and use RCP to manage the lifetime of such array class objects.

2 An example C++ program

The use of object-oriented (OO) programing in C++ is the major motivation for the development of RCP. OO programs are characterized by the use of abstract classes (i.e. interfaces) and concrete subclasses (i.e. implementations). In OO programs it is common that the selection of which concrete subclass(es) to use is not known until runtime. The "Abstract Factory" [2] is a popular design pattern that allows the flexible runtime selection of what concrete subclasses to create.

Below we describe a fictitious program that demonstrates some of the typical features of an OO program that uses dynamic memory management in C++. In this simple program, handling memory management using raw C++ pointers and calls to operator `new` and operator `delete` will appear fairly easy but larger more realistic OO programs are much more complicated and it is definitely not easy to do memory management without some help.

2.1 Example C++ program using raw dynamic memory management

One of the predominate features of this example program is the use of the following abstract interface base class `UtilityBase` that defines an interface to provide some useful capability.

```
  class UtilityBase {
  public:
    virtual void f() const = 0;
};
```

In our example program, `UtilityBase` will have two subclasses where one or the other will be used at runtime.

```
  class UtilityA : public UtilityBase {
  public:
    void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
  };

  class UtilityB : public UtilityBase {
  public:
    void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
  };
```

In this example program the above implementation functions just print to standard out.

Some of the clients in this program have to create `UtilityBase` objects without knowing exactly what concrete subclasses are being used. This is accomplished through the use of the "Abstract Factory" design pattern [2]. For `UtilityBase`, the abstract factory looks like

```
  class UtilityBaseFactory {
  public:
    virtual UtilityBase* createUtility() const = 0;
  };
```

and has the following factory subclasses for creating `UtilityA` and `UtilityB` objects.

```
class UtilityAFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityA(); }
};

class UtilityBFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityB(); }
};
```

Now let's assume that our example program has the following client classes.

```
// Simple client with no state
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};

// Client that maintains a pointer to a Utility object
class ClientB {
  UtilityBase *utility_;
public:
  ClientB() : utility_(0) {}
  ~ClientB() { delete utility_; }
  void initialize( UtilityBase *utility ) { utility_ = utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};

// Client that maintains pointers to UtilityFactory and Utility objects
class ClientC {
  const UtilityBaseFactory *utilityFactory_;
  UtilityBase *utility_;
  bool shareUtility_;
public:
  ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory),
    utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  ~ClientC() { delete utilityFactory_; delete utility_; }
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else  b->initialize(utilityFactory_->createUtility());
  }
};
```

The type of logic used in `ClientC` for determining when new objects should be created or when objects should be reused and passed around is common in larger more complicated OO programs.

The above client classes demonstrate two different types of associations between objects: *non-persisting* and *persisting*.

Non-Persisting associations exist only within a single function call and do not extend after the function has finished executing. For example, objects of type `ClientA` and `UtilityBase` have a non-persisting relationship through the function `ClientA::f(const UtilityBase &utility)`. Likewise, objects of type `ClientB` and `ClientA` have a non-persisting association through the function `ClientB::g( const ClientA &a )`.

Persisting associations are where a relationship between two objects exists past a single function call. The most typical kind of persisting association in an OO C++ program is where one object maintains a private pointer data member to another object. For example, persisting associations exist between a `ClientC` object, a `UtilityBaseFactory` and a `UtilityBase` object through the the private C++ pointer data members `ClientC::utilityFactory_` and `ClientC::utility_` respectively. Likewise, a persisting association exists between a `ClientB` object and a `UtilityBase` object through the private pointer data member `ClientB::utility_`.

Persisting relationships are significantly more complex than non-persisting relationships since a persisting relationship usually implies that some objects must be responsible for the lifetime of other objects. This is never the case in a non-persisting relationship as defined above.

Appendix E shows an example program that uses all of the C++ classes described above. The program in Appendix E has several memory management problems. An astute reader will notice that the `UtilityBaseFactory` created in `main()` gets deleted twice; once in the destructor for the `ClientC` object `c` and again at the end of `main()` in an explicit call to operator `delete`. This problem could be fixed in this program by arbitrating "ownership" of the `UtilityBaseFactory` object to either `main()` or the `ClientC` object, but not both which is the case in Appendix E.

A more difficult memory management problem to catch and fix occurs in the `ClientB` and `ClientC` objects regrading a shared `UtilityBase` object. When `shareUtility` is set to `false` (by the user in the commandline arguments) the objects `b1`, `b2` and `c` each own a pointer to different `UtilityBase` objects and the software will correctly delete each dynamically allocated object using one and only one call to operator `delete` (in the destructors of these classes). However, when `shareUtility` is to set to `true` the objects `b1`, `b2` and `c` will contain pointers to the same `UtilityBase` object and operator `delete` will be called on this shared `UtilityBase` object multiple times when `b1`, `b2` and `c` are destroyed. In this case, it is not so easy to arbitrate ownership of the shared `UtilityBase` object to the `ClientB` or the `ClientC` objects. Logic could be developed in this simple program to insure that ownership was assigned properly but such logic would enlarge the program, complicate maintenance, and would ultimately make the software components less reusable. In more complex programs, trying to dynamically arbitrate ownership at run time is much more difficult and error prone if done manually.

2.2 Refactored example C++ program using `Teuchos::RCP`

Now we describe how `RCP` can be used to greatly simplify dynamic memory management in these types of OO programs. Appendix F shows the refactoring of the program in Appendix E to use `RCP` for all persisting relationships. In general, refactoring software that uses raw C++ pointers to use `RCP` is as simple as replacing the type `T*` with `RCP<T>`, where `T` is nearly any class or built-in data type.

The first persisting relationship for which `RCP` is used is the relationship between a `Utility-`

BaseFactory object and a client that uses it. The refactoring changes the return type of Utility-BaseFactory::createUtility() from a raw UtilityBase* pointer to a RCP<UtilityBase> object. The new "Abstract Factory" class declarations (assuming that the symbols from the Teuchos namespace are in scope so that explicit Teuchos:: qualification is not necessary) become

```
class UtilityBaseFactory {
public:
  virtual RCP<UtilityBase> createUtility() const = 0;
};

class UtilityAFactory : public UtilityBaseFactory {
public:
  RCP<UtilityBase> createUtility() const { return rcp(new UtilityA()); }
};

class UtilityBFactory : public UtilityBaseFactory {
public:
  RCP<UtilityBase> createUtility() const { return rcp(new UtilityB()); }
};
```

In addition to the change of the return type, the refactoring also requires that calls to operator new be wrapped in calls to the templated function Teuchos::rcp(...).

The refactoring shown in Appendix F does not impact the definition of the class ClientA since this class does not have any persisting relationships with any other objects. However, the definitions of the classes ClientB and ClientC do change and become

```
class ClientB {
  RCP<UtilityBase> utility_;
public:
  void initialize(const RCP<UtilityBase> &utility) { utility_=utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};

class ClientC {
  RCP<UtilityBaseFactory> utilityFactory_;
  RCP<UtilityBase> utility_;
  bool shareUtility_;
public:
  ClientC( const RCP<UtilityBaseFactory> &utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory),
    utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  void h( const Ptr<ClientB> &b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else b->initialize(utilityFactory_->createUtility());
  }
};
```

The first thing that one should notice about the refactored ClientB and ClientC classes is that their destructors are gone. It turns out that the compiler-generated destructors do exactly the correct

thing (i.e. call the destructor on the RCP data members which in turns calls operator delete on the underlying reference-counted object when the reference count goes to zero). The second thing that one should notice is that the old default constructor ClientB::ClientB() which initialized the raw C++ pointer utility_ to null is no longer needed since RCP has a default constructor that does that. A third thing to notice about these refactored client classes is that the RCP objects are passed by const reference (see Appendix D) and not by value as the corresponding raw pointers where in the original unfactored classes. Passing RCP objects by const reference yields slightly more efficient code and simplifies stepping through the code in a debugger. For example, a function declared as

```
void someFunction( RCP<A> a );
```

will always result in the copy constructor for RCP being called (and therefore stepped into in a debugger) while this same function declared as:

```
void someFunction( const RCP<A> &a );
```

will often not require the copy constructor be called (except in cases where an implicit conversion is being performed as described in Appendix B) and thereby easing debugging.

Lastly, above, the class Ptr is a Teuchos non-reference-counted smart pointer class designed to avoid raw pointers. It is used for non-persisting associations where a raw pointer would otherwise be used. Ptr initializes to NULL and in debug mode it will throw exception exceptions when dereferencing NULL. Ptr plays a small role in the overall strategy to avoid all raw C++ pointers at the application programming level.

As an aside, note that Appendix D gives recommended idioms for how to pass raw C++ objects and RCP-wrapped objects to and from functions in a way that result in function prototypes becoming as self documenting as possible, help to avoid coding errors and increase the readability of C++ code. Also, in addition to the benefit that RCP eases dynamic memory management, the selective use of RCP and raw C++ object references extends the vocabulary of the C++ language by helping to distinguish between persisting and non-persisting associations. For example, when a one sees a function prototype where an object is passed through a RCP such as

```
class SomeClass {
public:
  void someFunction( const RCP<A> &a );
}
```

one can automatically deduce that “memory” of the A object will be retained (through a private RCP<A> data member in SomeClass no doubt) and that should automatically alter how the developer plans on calling that function and passing the A object. The refactored C++ program in Appendix F provides an example of how the idioms presented in Appendix D are put to use.

3 Additional and advanced features of RCP

The use cases for RCP described above comprise a large majority of the relavent use cases in most programs, but there there are some other use cases that require additional and more advanced features. Some of these additional features (the C++ declarations for which are shown in Appendix A) are mentioned below:

1. **Casting**

   RCP objects can be casted in a manner similar to casting raw C++ pointers and the same types of conversion rules apply. Analogs of the built-in casts `static_cast<>`, `const_cast<>` and `dynamic_cast<>` are supported by the non-member templated functions `rcp_static_cast<>`, `rcp_const_cast<>` and `rcp_dynamic_cast<>` respectively. See Appendix B for examples of how they are used.

2. **Reference-count information**

   The function `RCP::count()` returns the number of RCP objects that point to the underlying reference-counted object. This information can be useful in some cases.

3. **Customized deallocation policies**

   The default behavior of RCP is to call operator `delete` on reference-counted objects once the reference count goes to zero. While this is the most commonly needed behavior, there are use cases where more specialized dellocation polices are required. For these cases, there is an overloaded form of the templated function `Teuchos::rcp(...)` that takes a templated deallocation policy object that defines how a reference-counted object is deallocated when required.

4. **Associating extra data with a reference-counted object**

   There are some more difficult use cases where certain types of information or other objects must be bundled with a reference-counted object and must not be deleted until the reference-counted object is deleted. The non-member templated functions `set_extra_data<>(...)` and `get_extra_data<>(...)` serve this purpose (see item (6) in Appendix B). Note that the extra data mechanism relies on an `std::map` and string comparisons etc. and can impart some unacceptably high overhead in some use cases.

5. **Embeddeding an object on creation of an RCP object**

   Similar to the use of extra data, the RCP class also supports the concept of an embedded object. The functions `rcpWithEmbeddedObj[PreDestroy,PostDestroy](...)` (see 7 in Appendix B) can be used create an RCP object and embedd any other value-type object in the created RCPNode. This uses a customized deallocator class and imparts less overhead than the extra data feature at the cost of being less flexible (i.e. you can can only embedd a single value object and it must be done right when the first RCP object is created). The advantage of this approach is that access of the embedded object using the `get[Nonconst]EmbeddedObj(...)` is faster than when using extra data but requires that you provide more information.

6. **Checking for memory leaks from circular references**

In a debug build, the user can enable checking for memory leaks caused by circular references among RCP objects. If cicular references do exist, then `RCPNode` objects that where created but not removed are displaed at the end of a program. See the file `Teuchos_RCP.cpp` for details.

7. **Addressing circular references with weak RCP objects**

   The default mode for `RCP` is as a "strong" pointer. That means that the underlying reference-counted object is only deleted after all of the "strong" `RCP`s to the object are removed. Of course when you have a circular reference using "strong" `RCP`s, then that will never happen and a memory leak will be created for all of the objects involved in the cycle.

   To help address this problem, an `RCP` object can be tagged as a "weak" pointer. When all of the "strong" `RCP` objects goes away, the underlying reference-counted object is destroyed but the `RCPNode` object is not if there are any lingering (i.e. dangling) "weak" `RCP` objects. In a debug build of the code, all of the dangling "weak" `RCP` objects will thrown exceptions when clients try to dereference the object through the "weak" pointer. This functionality provides the foundation for a number of very advanced features. This capability imparts very little O(1) extra overhead.

   The ability to tag `RCP` objects as "weak" can be used to help address circular dependencies in general, clean, and safe way. In a debug build of the code, if any mistakes are made then exceptions will be thrown with an excellent error messages to help debug the problem. Without full blown garbage collection, this is about the best that we can do in C++.

4 Debugging C++ code

One issue that commonly comes up for beginning C++ programmers is how to debug programs that use RCP when they are just used to using raw C++ pointers. I am not going to cover the basics on how to use debuggers like GDB to debug C++ programs but I will give a few tips that should help beginning C++ programings get started.

This first problem that begining C++ programmers have is in trying to access the underlying raw C++ pointer in the debugger. For example, consider the function:

```
void someFunc( const RCP<const A> &a )
{
  a->f();
}
```

When debugging the above function in a debugger such as GDB, how can you access the underlying raw object of type A inside of the RCP<A> object a? Well, in GDB for instance, if you put a breakpoint on the line a->f(), you can print the address of the underlying object of type A by typing:

```
print a.ptr_
```

You can print the whole object with:

```
print *a.ptr_
```

and so on. In general, if you had code with a raw pointer named somePtr that was converted over to use RCP, you simply access the underlying raw C++ pointer using somePtr.ptr_ in the debugger. That is all there is to it. See the internal private reprsentation of RCP shown in Appendix A for more details.

One other note on debugging programs that use RCP that is worth mentioning is how to step through functions that take RCP as formal arguments. Consider the simple function someFunc(const RCP<const A>&) shown above. When this function is called by the following function:

```
void someFunc2( const RCP<const A> &a )
{
  someFunc(a);
}
```

you can step from the call someFunc(a) directly into someFunc(...) because the formal argument a of type const RCP<const A>& is a direct match.

However, if any implicit (or explicit) conversion of RCP objects is required to complete the function call, you will end up stepping into the copy constructor for RCP for each argument requiring a conversion. For example, the following functions require the copy constructor for RCP to be called in order to call and therefore step into the function someFunc(...):

```
void someFunc3( const RCP<A> &a )
{
  someFunc(a); // Convert from RCP<A> to RCP<const A>
}

void someFunc4( const RCP<const B> &b )
{
  someFunc(b); // Convert from RCP<const B> to RCP<const A>
}
```

To avoid having to step into the copy constructor for `RCP` in these cases, you can just directly set a breakpoint in the function `someFunc(...)`. In GDB you can do this by setting the breakpoint by typing:

```
break 'someFunc(
```

followed by typing `[Tab]` (which will expand the full function prototype) and then typing `[Enter]`. With this breakpoint set, you can just type `continue` in GDB and you will enter the function `someFunc(...)` without having to step through the copy constructors for `RCP`.

Debugging strategies in other debuggers is similar but you should get the idea.

5 Summary

The templated C++ class `RCP` provides a low-overhead option for (almost) automatic memory management in C++. This class has been developed and refined over many years and has been instrumental in improving the quality of software projects that use it consistently (for example see MOOCHO [1]). Careful use of `RCP` eliminates the need to manually call operator `delete` when dynamically allocated objects are no longer needed. Furthermore, it helps to reduce the amount of code that developers have to write. For example, most classes that use `RCP` for dynamically allocated memory do not need developer-supplied destructors. This because the compiler-generated destructors do the exactly correct thing which is to call destructors on an object's constituent data members. This was demonstrated in the difference between the original and refactored classes `ClientB` and `ClientC` described in Sections 2.1 and 2.2.

The class `RCP` also has advanced features not found in many other smart-pointer implementations such as the ability to attach extra data, the customization of the deallocation policy, cicular reference identification and debugging, and "weak" pointers to help resolve circular references.

References

[1] R. A. Bartlett. *MOOCHO : Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide*. Sandia National Labs, 2003.

[2] E. Gamma, R. Helm, R. Johnson, and John Vlissides. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[4] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[5] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[6] B. Stroustrup. *The C++ Programming Language, special edition*. Addison-Wesley, New York, 2000.

A C++ declarations for RCP

```
namespace Teuchos {

enum ENull { null };

enum EPrePostDestruction { PRE_DESTROY, POST_DESTROY };

template<class T>
class RCP {
public:
  typedef T  element_type;
  RCP( ENull null_arg = null );
  explicit RCP( T* p, bool has_ownership = false );
  template<class Dealloc_T>
  RCP( T* p, Dealloc_T dealloc, bool has_ownership );
  RCP(const RCP<T>& r_ptr);
  template<class T2> RCP(const RCP<T2>& r_ptr);
  ~RCP();
  RCP<T>& operator=(const RCP<T>& r_ptr);
  bool is_null() const;
  T* operator->() const;
  T& operator*() const;
  T* get() const;
  T* getRawPtr() const;
  Ptr<T> ptr() const;
  ERCPStrength strength() const;
  bool is_valid_ptr() const;
  int strong_count() const;
  int weak_count() const;
  int total_count() const;
  void set_has_ownership();
  bool has_ownership() const;
  Ptr<T> release();
  RCP<T> create_weak() const;
  template<class T2>
  bool shares_resource(const RCP<T2>& r_ptr) const;
  const RCP<T>& assert_not_null() const;
  const RCP<T>& assert_valid_ptr() const;
private:
  T *ptr_;
  RCPNode node_;
  ...
};

template<class T> RCP<T> rcp( T* p, bool owns_mem = true );

template<class T, class Dealloc_T> RCP<T> rcp( T* p, Dealloc_T dealloc,
  bool owns_mem );

template<class T> Teuchos::RCP<T> rcpFromRef( T& r );

template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObjPreDestroy( T* p, const Embedded &embedded,
  bool owns_mem = true );

template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObjPostDestroy( T* p, const Embedded &embedded,
  bool owns_mem = true );

template<class T, class Embedded>
RCP<T> rcpWithEmbeddedObj( T* p, const Embedded &embedded,
  bool owns_mem = true );

template<class T> bool is_null( const RCP<T> &p );
```

```
template<class T> bool operator==( const RCP<T> &p, ENull );

template<class T> bool operator!=( const RCP<T> &p, ENull );

template<class T1, class T2> bool operator==( const RCP<T1> &p1,
  const RCP<T2> &p2 );

template<class T1, class T2> bool operator!=( const RCP<T1> &p1,
  const RCP<T2> &p2 );

template<class T2, class T1> RCP<T2> rcp_implicit_cast(const RCP<T1>& p1);

template<class T2, class T1> RCP<T2> rcp_static_cast(const RCP<T1>& p1);

template<class T2, class T1> RCP<T2> rcp_const_cast(const RCP<T1>& p1);

template<class T2, class T1>
RCP<T2> rcp_dynamic_cast(const RCP<T1>& p1, bool throw_on_fail = false);

template<class T1, class T2>
void set_extra_data(const T1 &extra_data, const std::string& name,
  const Ptr<RCP<T2> > &p, EPrePostDestruction destroy_when = POST_DESTROY,
  bool force_unique = true );

template<class T1, class T2>
const T1& get_extra_data( const RCP<T2>& p, const std::string& name );

template<class T1, class T2>
T1& get_nonconst_extra_data( RCP<T2>& p, const std::string& name );

template<class T1, class T2>
Ptr<const T1> get_optional_extra_data( const RCP<T2>& p, const std::string& name );

template<class T1, class T2>
Ptr<T1> get_optional_nonconst_extra_data( RCP<T2>& p, const std::string& name );

template<class Dealloc_T, class T>
const Dealloc_T& get_dealloc( const RCP<T>& p );

template<class Dealloc_T, class T>
Dealloc_T& get_nonconst_dealloc( const RCP<T>& p );

template<class Dealloc_T, class T>
Ptr<const Dealloc_T> get_optional_dealloc( const RCP<T>& p );

template<class Dealloc_T, class T>
Ptr<Dealloc_T> get_optional_nonconst_dealloc( const RCP<T>& p );

template<class TOrig, class Embedded, class T>
const Embedded& getEmbeddedObj( const RCP<T>& p );

template<class TOrig, class Embedded, class T>
Embedded& getNonconstEmbeddedObj( const RCP<T>& p );

template<class TOrig, class Embedded, class T>
Ptr<const Embedded> getOptionalEmbeddedObj( const RCP<T>& p );

template<class TOrig, class Embedded, class T>
Ptr<Embedded> getOptionalNonconstEmbeddedObj( const RCP<T>& p );

template<class T>
std::ostream& operator<<( std::ostream& out, const RCP<T>& p );

} // namespace Teuchos
```

B RCP quick-start and reference

This appendix presents a short, but fairly comprehensive, quick-start for the use of RCP. The use cases described here should cover the overwhelming majority of the use instances of RCP in a typical program.

The following class hierarchy will be used in the C++ examples given below.

```
class A { public: virtual ~A(){} A& operator=(const A&){} virtual void f(){} };
class B1 : virtual public A {};
class B2 : virtual public A {};
class C : virtual public B1, virtual public B2 {};

class D {};
class E : public D {};
```

All of the following code examples used in this appendix are assumed to be in the namespace Teuchos or have appropriate `using Teuchos::...` declarations. This removes the need to explicitly use `Teuchos::` to qualify classes, functions and other declarations from the Teuchos namespace. Note that some of the runtime checks are denoted as "debug runtime checked" which means that checking will only be performed in a debug build (that is one where the macro _DEBUG is defined at compile time).

1. **Creation of RCP objects**

   (a) **Initializing a RCP object to NULL**

   ```
   RCP<C> c_ptr;
   ```

   or

   ```
   RCP<C> c_ptr = null;
   ```

   (b) **Creating a RCP object using new**

   ```
   RCP<C> c_ptr = rcp(new C);
   ```

   or

   ```
   RCP<C> c_ptr(new C);
   ```

   NOTE: Prefer to define and use non-member constructor functions that yeild:

   ```
   RCP<C> c_ptr = newC();
   ```

   (c) **Creating a RCP object to an array allocated using new[n]**
   See the class `Teuchos::ArrayRCP` and the function `arcp<T>(int n)`.

   (d) **Initializing a RCP object to an object <u>not</u> allocated with new**

   ```
   C c;
   RCP<C> c_ptr = rcpFromRef(c);
   ```

   (e) **Copy constructor (implicit casting)**

```
RCP<C>       c_ptr  = rcp(new C); // No cast
RCP<A>       a_ptr  = c_ptr;      // Cast to base class
RCP<const A> ca_ptr = a_ptr;      // Cast from non-const to const
```

(f) **Representing constantness and non-constantness**

i. **Non-constant pointer to non-constant object**

```
RCP<C> c_ptr;
```

ii. **Constant pointer to non-constant object**

```
const RCP<C> c_ptr;
```

iii. **Non-Constant pointer to constant object**

```
RCP<const C> c_ptr;
```

iv. **Constant pointer to constant object**

```
const RCP<const C> c_ptr;
```

2. **Reinitialization of `RCP` objects (using assignment operator)**

(a) **Resetting from a raw pointer**

```
RCP<A> a_ptr;
a_ptr = rcp(new A());
```

(b) **Resetting to null**

```
RCP<A> a_ptr = rcp(new A());
a_ptr = null; // The A object will be deleted here
```

(c) **Assigning from a `RCP` object**

```
RCP<A> a_ptr1;
RCP<A> a_ptr2 = rcp(new A());
a_ptr1 = a_ptr2; // Now a_ptr1 and a_ptr2 point to same A object
```

3. **Accessing the reference-counted object**

(a) **Access to object reference (debug runtime checked)**

```
C &c_ref = *c_ptr;
```

(b) **Access to object pointer (unchecked, may return `NULL`, NOT RECOMMENDED)**

```
C *c_rptr = c_ptr.get();
```

WARNING: Avoid exposing raw C++ pointers in your program!

(c) **Access to object pointer (debug runtime checked, will not return `NULL`, NOT RECOMMENDED)**

```
C *c_rptr = &*c_ptr;
```

WARNING: Avoid exposing raw C++ pointers in your program!

(d) **Access of object's member (debug runtime checked)**

```
c_ptr->f();
```

(e) **Testing for non-null**

```
if (!is_null(a_ptr)) std::cout << "a_ptr is not null!\n";
```

or

```
if (a_ptr != null) std::cout << "a_ptr is not null!\n";
```

(f) **Testing for null**

```
if (is_null(a_ptr)) std::cout << "a_ptr is null!\n";
```

or

```
if (a_ptr == null) std::cout << "a_ptr is null!\n";
```

4. **Casting**

(a) **Implicit casting (see copy constructor above)**

i. **Using copy constructor (see above)**

ii. **Using conversion function**

```
RCP<C>       c_ptr  = rcp(new C);                       // No cast
RCP<A>       a_ptr  = rcp_implicit_cast<A>(c_ptr);      // To base
RCP<const A> ca_ptr = rcp_implicit_cast<const A>(a_ptr);// To const
```

(b) **Casting away `const`**

```
RCP<const A>  ca_ptr = rcp(new C);
RCP<A>        a_ptr  = rcp_const_cast<A>(ca_ptr); // cast away const!
```

(c) **Static cast (no runtime check)**

```
RCP<D>  d_ptr = rcp(new E);
RCP<E>  e_ptr = rcp_static_cast<E>(d_ptr); // Unchecked, unsafe?
```

(d) **Dynamic cast (runtime checked, failed cast allowed)**

```
RCP<A>  a_ptr  = rcp(new C);
RCP<B1> b1_ptr = rcp_dynamic_cast<B1>(a_ptr);  // Checked, safe!
RCP<B2> b2_ptr = rcp_dynamic_cast<B2>(b1_ptr); // Checked, safe!
RCP<C>  c_ptr  = rcp_dynamic_cast<C>(b2_ptr);  // Checked, safe!
```

(e) **Dynamic cast (runtime checked, failed cast not allowed)**

```
RCP<A>  a_ptr1  = rcp(new C);
RCP<A>  a_ptr2  = rcp(new A);
RCP<B1> b1_ptr1 = rcp_dynamic_cast<B1>(a_ptr1,true); // Success!
RCP<B1> b1_ptr2 = rcp_dynamic_cast<B1>(a_ptr2,true); // Throw std::bad_cast!
```

5. **Customized deallocators**

   (a) **Creating a RCP object with a custom deallocator**

   ```
   RCP<C> c_ptr = rcp(new C[N],DeallocArrayDelete<C>(),true);
   ```

   (b) **Access customized deallocator (runtime checked, throws on failure)**

   ```
   const DeallocArrayDelete<C>
     &dealloc = get_dealloc<DeallocArrayDelete<C> >(c_ptr);
   ```

   (c) **Access optional customized deallocator**

   ```
   Ptr<const DeallocArrayDelete<C> >
     dealloc = get_optional_dealloc<DeallocArrayDelete<C> >(c_ptr);
   if (!is_null(dealloc))
     std::cout << "This deallocator exits!\n";
   ```

6. **Managing extra data**

   (a) **Adding extra data (post-destruction of extra data)**

   ```
   set_extra_data(rcp(new B1), "A:B1", inOutArg(a_ptr));
   ```

   (b) **Adding extra data (pre-destruction of extra data)**

   ```
   set_extra_data(rcp(new B1), "A:B1", inOutArg(a_ptr), PRE_DESTORY);
   ```

   (c) **Retrieving extra data**

   ```
   get_extra_data<RCP<B1> >(a_ptr,"A:B1")->f();
   ```

   (d) **Resetting extra data**

   ```
   get_extra_data<RCP<B1> >(a_ptr,"A:B1") = rcp(new C);
   ```

   (e) **Retrieving optional extra data**

   ```
   Ptr<const RCP<B1> > b1 =
     get_optional_extra_data<RCP<B1> >(a_ptr,"A:B1");
   if (!is_null(b1))
     (*b1)->f();
   ```

7. **Embedded objects**

   (a) **Creating an RCP object with embedded data**

   ```
   RCP<D> d_ptr(new D);
   RCP<A> a_ptr rcpWithEmbeddedObj(new C, rcp(new D));
   ```

   (b) **Extract reference to const embedded object**

   ```
   const RCP<D> &d_ptr = getEmbeddedObj<C,RCP<D> >(a_ptr);
   ```

   (c) **Extract reference to nonconst embedded object**

   ```
   RCP<D> &d_ptr = getNonconstEmbeddedObj<C,RCP<D> >(a_ptr);
   d_ptr = null; // Sets the actual embedded RCP<D> object in a_ptr to null!
   ```

C Commandments for the use of RCP

Here are listed commandments for the use of RCP. These commandments reinforce some of the material in the quick-start in Appendix B. Along with each commandment is one or more anti-commandments stating the negative of the commandment. C++ code fragments are also included to demonstrate each commandment and anti-commandment.

Commandment 1 *Thou shall put a pointer for an object allocated with operator* `new` *into a* `RCP` *object only once. The best way to insure this is to call operator* `new` *directly in a call to* `rcp(...)` *to create a dynamically allocated object that is to be managed by a* `RCP` *object. Better yet, define and use non-member constructor functions and never use raw calls to* `new` *at the application programming level See item (1b) in Appendix B.*

Anti-Commandment 1 *Thou shall never give a raw C++ pointer returned from operator* `new` *to more than one* `RCP` *object.*

Example:

```
A *ra_ptr = new C;
RCP<A> a_ptr1 = rcp(ra_ptr); // Okay
RCP<A> a_ptr2 = rcp(ra_ptr); // no, No, NO !!!!
```

Anti-Commandment 2 *Thou shall never give a raw C++ pointer to an array of objects returned from operator* `new[]` *to a* `RCP` *object using* `rcp(new C[n])`.

Example:

```
RCP<std::vector<C> > c_array_ptr1 = rcp(new std::vector<C>(N)); // Okay
RCP<C> c_array_ptr3 = rcp(new C[n]);  // no, No, NO!
```

Commandment 2 *Thou shall only create a* `NULL RCP` *object by using the default constructor or by using the* `null` *enum (and its associated special constructor) (see item (1a) in Appendix B). Trying to assign to* `NULL` *or* `0` *will not compile.*

Anti-Commandment 3 *Thou shall not create a* `NULL RCP` *object using the templated function* `rcp(...)` *since it is very verbose and complicates maintenance.*

Example:

```
RCP<A> a_ptr1 = null;          // Yes :-)
RCP<A> a_ptr2 = rcp<A>(NULL);  // No, too verbose :-(
```

Commandment 3 *Thou shall only pass a raw pointer for an object that is not allocated by operator* `new` *(e.g. allocated on the stack) into a* `RCP` *object by using the templated function* `rcpFromRef<T>(T& t)` *described in Appendix B.*

Anti-Commandment 4 *Thou shall never pass a pointer for an object not allocated with operator* `new` *into a* `RCP` *object without setting* `owns_mem` *to* `false`*.*

Example:

```
C c;
RCP<A> a_ptr1 = rcpFromRef(c);  // Yes :-)
RCP<A> a_ptr2 = rcp(&c);        // no, No, NO !!!!
```

Commandment 4 *Thou shall only cast between* `RCP` *objects using the default copy constructor (for implicit conversions) and the nonmember template functions* `rcp_implicit_cast<>(...)`, `rcp_static_cast<>(...)`, `rcp_const_cast<>(...)` *and* `rcp_dynamic_cast<>(...)` *(see item (4) in Appendix B).*

Anti-Commandment 5 *Thou shall never convert between* `RCP` *objects using raw pointer access.*

Example:

```
RCP<A>     a_ptr   = rcp(new C);
RCP<B1>    b1_ptr1 = rcp_dynamic_cast<B1>(a_ptr);         // Yes :-)
RCP<B1>    b1_ptr2 = rcp(dynamic_cast<B1*>(a_ptr.get())); // no, No, NO !!!
```

D Recommendations for passing objects to and from C++ functions

Below are recommended idioms for passing required[1] and optional[2] arguments into and out of C++ functions for various use cases and different types of objects. These idioms show how to write function prototype argument declarations which exploit the C++ language in a way that makes these function prototypes as self documenting as possible, avoids coding errors, and increases readability[3] of C++ code. In general, `RCP<T>` objects should be passed and manipulated as though they where raw C++ pointer `T*` objects. However, while raw C++ pointer objects should generally be passed by value, `RCP` objects should generally be passed by reference to avoid unecessary copy constructor calls.

Argument purpose	Non-Persisting	Persisting

non-changeable object (required[1])	`S s`
	or
	`const S s`
	or
	`const S &s`
		`const RCP<const S> &s`

non-changeable object (optional[2])	`const Ptr<const S> &s`	`const RCP<const S> &s`
changeable object	`const Ptr<S> &s`	`const RCP<S> &s`

C++ declarations for passing small concrete objects (i.e. with value semantics) to and from functions where `S` is a place holder for an actual built-in or user-defined data type.

Argument purpose	Non-Persisting	Persisting
non-changeable object (required[1])	`const A &a`	`const RCP<const A> &a`
non-changeable object (optional[2])	`const Ptr<const A> &a`	`const RCP<const A> &a`
changeable object	`const Ptr<A> &a`	`const RCP<A> &a`

C++ declarations for passing abstract objects (i.e. with reference or pointer semantics) or large concrete objects (i.e. that are too expensive to copy) to and from functions where `A` is a place holder for an actual (abstract) C++ base class.

[1] Required arguments must be bound to valid objects (i.e. can not be `NULL`)

[2] Optional arguments may be `NULL` in some cases

[3] What makes code more "readable" is subjective of course.

E Listing: Example C++ program using raw dynamic memory management

```
#include "example_get_args.hpp"

// Abstract interfaces
class UtilityBase {
public:
  virtual void f() const = 0;
};
class UtilityBaseFactory {
public:
  virtual UtilityBase* createUtility() const = 0;
};

// Concrete implementations
class UtilityA : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
};
class UtilityB : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
};
class UtilityAFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityA(); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
  UtilityBase* createUtility() const { return new UtilityB(); }
};

// Client classes
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};
class ClientB {
  UtilityBase *utility_;
public:
  ClientB() : utility_(0) {}
  ~ClientB() { delete utility_; }
  void initialize( UtilityBase *utility ) { utility_ = utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};
class ClientC {
  const UtilityBaseFactory *utilityFactory_;
  UtilityBase *utility_;
  bool shareUtility_;
public:
  ClientC( const UtilityBaseFactory *utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
```

```
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  ~ClientC() { delete utilityFactory_; delete utility_; }
  void h( ClientB *b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else b->initialize(utilityFactory_->createUtility());
  }
};

// Main program
int main( int argc, char* argv[] )
{
  // Read options from the commandline
  bool useA, shareUtility;
  example_get_args(argc,argv,&useA,&shareUtility);
  // Create factory
  UtilityBaseFactory *utilityFactory = 0;
  if(useA) utilityFactory = new UtilityAFactory();
  else     utilityFactory = new UtilityBFactory();
  // Create clients
  ClientA a;
  ClientB b1, b2;
  ClientC c(utilityFactory,shareUtility);
  // Do some stuff
  c.h(&b1);
  c.h(&b2);
  b1.g(a);
  b2.g(a);
  // Cleanup memory
  delete utilityFactory;
}
```

F Listing: Refactored example C++ program using RCP

```
#include "Teuchos_RCP.hpp"
#include "example_get_args.hpp"

// Inject symbols for RCP so we don't need Teuchos:: qualification
using Teuchos::RCP;
using Teuchos::rcp; // Warning! This can be dangerous and is not to be used in general!
using Teuchos::Ptr;

// Abstract interfaces
class UtilityBase {
public:
  virtual void f() const = 0;
};
class UtilityBaseFactory {
public:
  virtual RCP<UtilityBase> createUtility() const = 0;
};

// Concrete implementations
class UtilityA : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityA::f() called, this="<<this<<"\n"; }
};
class UtilityB : public UtilityBase {
public:
  void f() const { std::cout<<"\nUtilityB::f() called, this="<<this<<"\n"; }
};
class UtilityAFactory : public UtilityBaseFactory {
public:
  RCP<UtilityBase> createUtility() const { return rcp(new UtilityA()); }
};
class UtilityBFactory : public UtilityBaseFactory {
public:
  RCP<UtilityBase> createUtility() const { return rcp(new UtilityB()); }
};

// Client classes
class ClientA {
public:
  void f( const UtilityBase &utility ) const { utility.f(); }
};
class ClientB {
  RCP<UtilityBase> utility_;
public:
  void initialize(const RCP<UtilityBase> &utility) { utility_=utility; }
  void g( const ClientA &a ) { a.f(*utility_); }
};
class ClientC {
  RCP<const UtilityBaseFactory> utilityFactory_;
  RCP<UtilityBase> utility_;
  bool shareUtility_;
```

```
public:
  ClientC( const RCP<const UtilityBaseFactory> &utilityFactory, bool shareUtility )
    :utilityFactory_(utilityFactory)
    ,utility_(utilityFactory->createUtility())
    ,shareUtility_(shareUtility) {}
  void h( const Ptr<ClientB> &b ) {
    if( shareUtility_ ) b->initialize(utility_);
    else b->initialize(utilityFactory_->createUtility());
  }
};

// Main program
int main( int argc, char* argv[] )
{
  // Read options from the commandline
  bool useA, shareUtility;
  example_get_args(argc,argv,&useA,&shareUtility);
  // Create factory
  RCP<UtilityBaseFactory> utilityFactory;
  if(useA) utilityFactory = rcp(new UtilityAFactory());
  else     utilityFactory = rcp(new UtilityBFactory());
  // Create clients
  ClientA a;
  ClientB b1, b2;
  ClientC c(utilityFactory,shareUtility);
  // Do some stuff
  c.h(&b1);
  c.h(&b2);
  b1.g(a);
  b2.g(a);
}
```

Sandia National Laboratories